



**DEPARTAMENTO
DE COMPUTACION**

Facultad de Ciencias Exactas y Naturales - UBA

TP1

Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Sebastián Fernandez Ledesma	392/06	sfernandezledesma@gmail.com
Fernando Gasperi Jabalera	56/09	fgasperijabalera@gmail.com
Maximiliano Wortman	892/10	maxifwortman@gmail.com
Santiago Camacho	110/09	santicamacho90@gmail.com



Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (54 11) 4576-3359

<http://www.fcen.uba.ar>

Índice

1. Problema 1: Puentes sobre lava caliente	3
1.1. Presentación del problema	3
1.2. Resolución	3
1.2.1. Algoritmo	3
1.2.2. Pseudocódigo	4
1.3. Demostración	4
1.4. Análisis de complejidad	6
1.5. Test de complejidad	7
1.6. Testing	7
2. Problema 2: Horizontes lejanos	8
2.1. Presentación del problema	8
2.2. Resolución	8
2.2.1. Algoritmo	8
2.2.2. Pseudocódigo	9
2.3. Demostración	9
2.4. Análisis de complejidad	9
2.5. Test de complejidad	9
2.6. Testing	9
3. Problema 3: Biohazard	10
3.1. Presentación del problema	10
3.2. Resolución	10
3.2.1. Algoritmo	10
3.2.2. Pseudocódigo	11
3.3. Demostración	11
3.4. Análisis de complejidad	11
3.5. Test de complejidad	11
3.6. Testing	11

1. Problema 1: Puentes sobre lava caliente

1.1. Presentación del problema

Se quiere atravesar un puente con n tablones dando saltos acotados por un valor de x tablones. Se empieza afuera del puente y se pretende salir completamente de éste, es decir que como mínimo hay que saltar una vez (en el caso trivial de que $x > n$). La dificultad consiste en que ciertos tablones conocidos están rotos, y no pueden ser pisados. Lo que pide el problema es minimizar la cantidad de saltos para atravesar el puente, o aclarar que es imposible. Los puentes estarán definidos como $t_1 t_2 \dots t_n$ donde $t_i = 0$ si el tablón está sano o $t_i = 1$ si está dañado.

Por ejemplo, podríamos tener el puente 0 1 0 0 con un salto máximo igual a 2. Como se arranca afuera, saltar al primer tablón se considera como un salto de 1 tablón. En este caso no podemos saltar los dos tablones permitidos porque el segundo tablón está roto (el puente, usando X para marcar donde estamos parados, se vería así: X 1 0 0). El segundo salto sí podremos saltar los 2 tablones, quedando 0 1 X 0, y con el tercer salto saldremos del puente.

Una configuración más complicada podría ser el puente 0 0 1 0 0 0 1 1 0 0 para un salto máximo de 3 tablones, ya que ahora tenemos dos posibilidades: saltar al primer o al segundo tablón. Usaremos un algoritmo goloso para resolver el problema (saltar la mayor cantidad posible de tablones) y demostraremos que es correcto y que es la solución óptima para el problema.

1.2. Resolución

1.2.1. Algoritmo

Dado este problema de optimización planteamos resolverlo con un algoritmo goloso, que consiste en seguir "una heurística consistente para elegir la opción óptima en cada paso local con la esperanza de llegar a una solución general óptima" [Cormen p.414 (Greedy Algorithms)]. El problema a optimizar es encontrar la mínima cantidad de saltos para cruzar el puente, y la decisión golosa o la opción óptima en cada paso local es elegir el tablon más lejano que pueda alcanzar el participante de acuerdo al rango de salto que tenga.

El algoritmo recibe un vector con los tablones del puente (`puente[i]`) y un entero que representa el máximo salto que puede dar el participante (`maxSalto`).

Teniendo esa información inicializamos la variable *actual* y *proximo* en 0, que son enteros. La primera representa en que posición del puente se ubica el participante y la segunda la posición del salto más lejano que puede alcanzar a un tablon. Estas variables son actualizadas por un ciclo, que en el caso que haya solución corre hasta que la posición *actual* sea mayor a la cantidad de tablones, es decir que el participante haya cruzado el puente.

Dentro del ciclo, se calcula la variable *proximo* con una función (*calcularProximoTablon*) que recibe el *puente* la posición *actual* y el *maxSalto* y prueba desde el salto más largo que puede dar hasta el mínimo cual es el próximo tablon óptimo, si no existe, entonces devuelve una excepción y hace que el algoritmo termine o en caso contrario el ciclo lo guarda en un vector de *saltos*. *Actual* se actualiza a la posición *proximo* en cada iteración que significa que el participante avanza en cada vuelta del ciclo.

Una vez que termina el ciclo el algoritmo devuelve el arreglo de *saltos*, que es vacío si no existe solución.

1.2.2. Pseudocódigo

Algorithm 1 $\text{cruzarPuede}(\text{vector<int> puente, int maxSalto}) \rightarrow \text{vector<int> saltos}$

```

int cantidadTablones  $\leftarrow |puente| - 2$  // El vector tiene dos tablones más: tanto el primero
como el último se consideran fuera del puente
int actual  $\leftarrow 0$ 
int proximo  $\leftarrow 0$ 
while actual  $\leq$  cantidadTablones do
  proximo  $\leftarrow$  calcularProximoTablon(puente, actual, maxSalto)
  if proximo == -1 then
    return vector vacío
  end if
  introducirAlFinal(saltos, proximo)
  if proximo > cantidadTablones then
    return saltos
  end if
  actual  $\leftarrow$  proximo
end while

```

Algorithm 2 $\text{int calcularProximoTablon}(\text{vector<int> puente, int actual, int maxSalto})$

```

int cantidadTablones  $\leftarrow |puente| - 2$ 
while maxSalto > 0 do
  if actual + maxSalto > cantidadDeTablones then
    return cantidadDeTablones + 1
  end if
  if puente[actual + maxSalto] == 1 then
    return actual + maxSalto
  end if
  maxSalto  $\leftarrow$  maxSalto - 1
end while
return -1

```

1.3. Demostración

Vamos a tratar de probar que dado una secuencia de saltos, si para cada salto s , s es un "salto máximo" si la sumatoria de saltos es mayor a la cantidad de tablones del puente, entonces nuestra secuencia es solución del problema.

Dada un $\text{Sec<Salto> } se$.

$$(\forall i : \text{Nat}, i < se.\text{long})(esMax(se_i) \wedge \sum_{j=0}^{se.\text{long}-1} se_j = \text{puente}.\text{long}) \implies \\ \nexists (se' : \text{Sec} < \text{Salto} >) / se.\text{long} < se'.\text{long} \wedge \sum_{j=0}^{se'.\text{long}-1} se'_j \geq \text{puente}.\text{long})$$

Llamemos a $sMax$ a la secuencia de saltos máximos obtenida.

Primero veamos que si existe una secuencia de saltos maximos, es unica. Esto es fácil de ver ya que, supongamos que existe otra secuencia s distinta de saltos maximos. Bueno, me paro en el primer salto s_i tal que s_i es distinto de $sMax_i$.

Entonces:

si $s_i < sMax_i \implies s_i$ no es máximo, así que s no es una secuencia de saltos máximos.

si $s_i > sMax_i \implies sMax_i$ no es máximo lo cual es absurdo.

Ahora veamos que dado una secuencia s y un salto s_i , ($i: \text{Nat}$, $i < s.long$ y una posición ($p: \text{Nat}$, $p = \sum_{j=0}^{i-1} s_j$), yo puedo "maximizar" ese salto, y esto no me agrega elementos a s y me mantiene la distancia recorrida por esa secuencia, donde maximizar un salto significa remplazarlo por el salto mas largo que puede hacerse desde la posición p en la que esta parado y la distancia recorrida por una secuencia s es $\sum_{j=0}^{s.long-1} s_j$

Llamemos m_p :Salto, al salto máximo que podemos hacer desde la posición p . m_p no puede ser menor a s_i ya que no seria máximo, si m_p fuera igual no agregaria saltos a la secuencia s . Ahora veamos que si el caso es que $m_p > s_i$ entonces para los s_j , ($j: \text{Nat}$, $i < j < s.long$) que se solapen con m_p , sucede que si $s_j + s_i < m_p \implies$ borro a s_j lo cual me saca un salto (Si fueran más de uno, sacaría más de uno) .

Ahora si $s_j + s_i > m_p \wedge m_p > s_i \implies$ reemplazo a s_j por $s_j - m_p + s_i$, y la secuencia me queda igual. Por lo tanto dada cualquier secuencia s , $\text{maximizar}(s_i)$ y obtener la nueva secuencia $s' \implies s.long \geq s'.long$.

Bueno y ahora, supongamos que yo tengo una secuencia s que es solución del problema. Si yo maximizo cada salto de s , a partir de la posición 0, obtengo una secuencia de saltos maximos que tiene igual cantidad de elementos que s y todos los saltos son máximos. Como todos los saltos son máximos esa secuencia es una secuencia de saltos máximos la cual habiamos llamado a $sMax$.

Y como $sMax$ es única vemos que $sMax$ tiene la misma o menor cantidad de elementos que s y recorre la misma distancia. Como s es solución, no puede existir una secuencia con menos elementos y igual o mayor distancia recorrida. Por lo tanto $sMax$ tiene la misma cantidad de elementos que s y es solución.

LIMPIADA DE CARA FER

Definimos un salto s como un natural mayor a 0 y menor o igual a la distancia máxima que puede recorrer el participante, de sólo un salto, medida en tableros

$$s \in \text{Saltos} \Leftrightarrow (s \in \mathbb{N}_{>0} \wedge s \leq \text{dist}_{max})$$

Definimos un puente como una función $p: \mathbb{N}_{>0} \rightarrow \mathbb{N}$

$$p(i) = \begin{cases} 1 & i \leq 0 \\ 0 & i > \# \text{tableros} \\ 0 & i > 0 \wedge i \leq \# \text{tableros} \wedge i \in \text{Tableros} \\ 1 & i > 0 \wedge i \leq \# \text{tableros} \wedge i \notin \text{Tableros} \end{cases}$$

Para cada posición i del puente definimos su salto máximo s_{max} como

$$s_{max} = \max\{n \in \mathbb{N}_{>0} \mid n \leq dist_{max} \wedge \neg p(n)\}$$

Nuestra implementacion recorre el puente dando saltos, garantizando que en cada salto, la distancia recorrida es máxima. Es decir, no existe otro salto tal que la distancia desde donde estamos parados es mayor a la del salto actual y el tablón en el que caes no esta roto. *Distancia* es un $\text{Nat} > 0$.

Salto es Nat tal que $\forall s : \text{Salto}, s > 0 \wedge s \leq \text{Distancia}$

Vamos a tratar de probar que dado una secuencia de saltos, si para cada salto s , s es un "salto maximo" si la sumatoria de saltos es mayor a la cantidad de tablonos del puente, entonces nuestra secuencia es solucion del problema.

Dada un $\text{Sec} < \text{Salto} > se$.

$$(\forall i : \text{Nat}, i < se.long)(esMax(se_i) \wedge \sum_{j=0}^{se.long-1} se_j = puente.long) \implies \\ \exists (se' : \text{Sec} < \text{Salto} >) / se.long < se'.long \wedge \sum_{j=0}^{se'.long-1} se'_j \geq puente.long //$$

Llamemos a $sMax$ a la secuencia de saltos maximos obtenida. Supongamos que existe secuencia s de saltos tal que la cantidad de elementos de s es menor a $sMax$ y la sumatoria de saltos es igual o mayor a la cantidad de tablonos. Bueno en particular, existe al menos un salto s_i , tal que s_i , es mayor a $sMax_i$, ya que si todos los s_i , son menores a su correspondiente $sMax_i$, entonces la sumatoria de $sMax$ es mayor que la sumatoria de s . (comprobar esto ad-hoc, probablemente sale por induccion). Bueno, supongamos que agarro el primero de todos los s_i , que es mas grande que su correspondiente $sMax_i$. Hasta ese momento las dos subsecuencias (desde el principio hasta el elemento i) pesan lo mismo, entonces s_i esta parado en el mismo lugar y hace un salto mas grande que el salto maximo ($sMax_i$), lo cual es absurdo. Por lo tanto queda comprobado que ese s_i , no puede existir y la solucion es máxima.

1.4. Análisis de complejidad

El algoritmo *cruzarPuentes* se puede dividir en dos partes:

- Inicialización
- Ciclo

En la Inicialización el algoritmo asigna 3 variables en $O(1)$, con lo cual su complejidad es $3 * O(1) = O(1)$ El ciclo, como peor caso, itera hasta n veces, donde n es la *cantidadDeTablonos*. Adentro del ciclo se calcula la función *calcularProximoTablon* para cada iteración. Esta nos dice el índice del próximo tablón óptimo para saltar, y a su vez es otro ciclo que se repite k veces haciendo una cantidad acotada de operaciones $O(1)$, donde k es la variable de entrada *maxsalto* que es un valor acotado. Luego el ciclo continua haciendo asignaciones y condicionales y devoluciones en $O(1)$.

Haciendo el calculo de complejidad obtenemos:

$$O(cruzarPuentes) = 3 * O(1) + n(O(k))$$

Que es lo mismo que:

$$O(cruzarPuentes) = O(n * k)$$

Podemos ver que el algoritmo depende de k , es decir, del *maxsalto* del participante, con lo cual podemos considerar que tiene una complejidad pseudopolinomial ya que depende de una variable de entrada, pero como sabemos que k es acotado por una constante finalmente podemos concluir que es de orden $O(n)$.

1.5. Test de complejidad

1.6. Testing

2. Problema 2: Horizontes lejanos

2.1. Presentación del problema

Dado un conjunto de rectángulos en un plano, todos apoyados sobre una línea recta horizontal, como en las siguientes figuras, se pide eliminar las líneas que colisionen con algún otro rectángulo, donde colisionar también es solamente “tocar” otra línea.

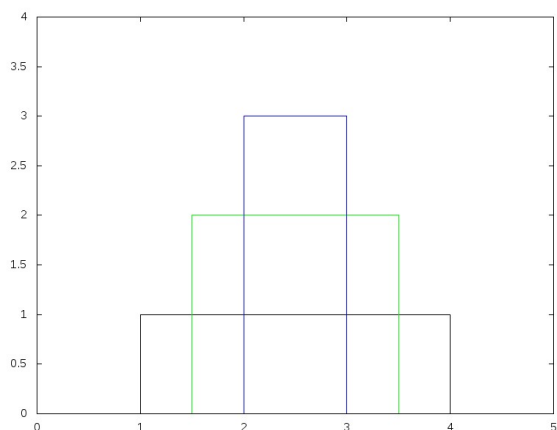


Figura 1: Con colisión total

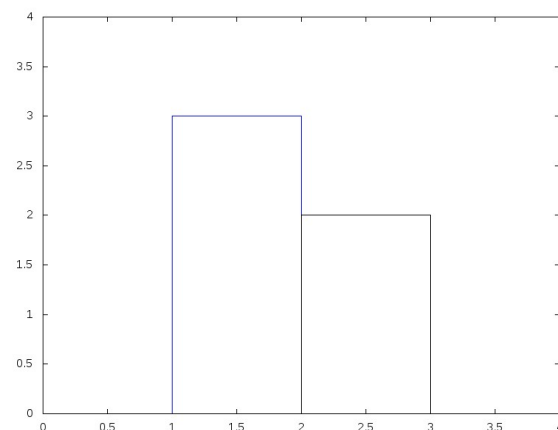


Figura 2: Sólo se tocan los bordes

Así, tras ejecutar el algoritmo, el resultado para los ejemplos anteriores sería:

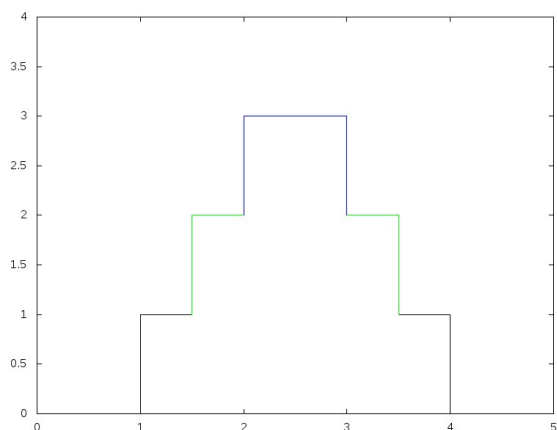


Figura 3: Resultado con colisión total

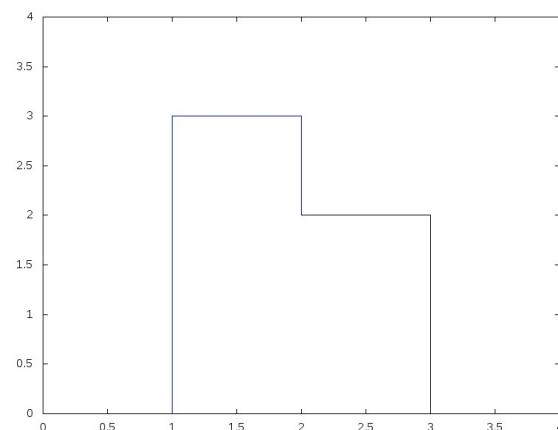


Figura 4: Resultado cuando sólo se tocan los bordes

Como requerimiento adicional, el algoritmo para n rectángulos debe tener una complejidad temporal estrictamente menor que $O(n^2)$.

2.2. Resolución

2.2.1. Algoritmo

COMPLETAR

2.2.2. Pseudocódigo

```
input: edificios
while quedan edificios do
  if empieza edificio then
    registro el edificio como abierto
    if altura del edificio es mayor a la del contorno then
      agrego la altura del edificio al contorno
    end if
  else
    saco al edificio de los abiertos
    if este edificio le daba la altura al contorno then
      agrego la altura del edificio abierto que le siga en altura al contorno
    end if
  end if
end while
return contorno
```

2.3. Demostración

2.4. Análisis de complejidad

2.5. Test de complejidad

2.6. Testing

3. Problema 3: Biohazard

3.1. Presentación del problema

Tenemos una cantidad n de productos químicos que deseamos repartir en la menor cantidad de camiones posible, con la particularidad de que no se supere un umbral M de peligrosidad, el cual tenemos definido para cada par de productos. Entonces, la peligrosidad para un camión C está dada por

$$\text{peligrosidadCamion}(C) = \sum_{\substack{p_i, p_j \in C \\ i < j}} \text{peligrosidadEntreProductos}(p_i, p_j)$$

Resolveremos este problema usando un algoritmo de backtracking implementando podas para reducir la complejidad temporal.

3.2. Resolución

3.2.1. Algoritmo

Definimos la estructura Camión que usaremos como contenedor de los productos como

$$\text{Camion es tupla } \langle \text{peligrosidad} : \text{nat}, \text{productos} : \text{conjunto}(\text{nat}) \rangle$$

La clave está en la función de backtracking

$$bt(p : \text{nat}, \text{camiones} : \text{lista} \langle \text{Camion} \rangle, \text{mejorTemp} : \text{lista} \langle \text{Camion} \rangle, M : \text{nat}, n : \text{nat})$$

donde p es el i -ésimo producto a repartir, camiones es la lista actual de camiones (esto se pasa por referencia), mejorTemp es la mejor solución hasta el momento (también se pasa por referencia), M es la máxima peligrosidad permitida por camión, y n es la cantidad de productos. Dados los parámetros, lo que hace es lo siguiente:

- Si ya se introdujeron todos los productos ($p = n$), salgo, pero si además la lista de camiones es más chica que la mejor solución temporal, la pongo como mejor solución.
- Para cada camión en camiones , intento agregar a p . Si puedo hacerlo, llamo a $bt(p+1, \text{camiones}, \text{mejorTemp}, M, n)$, e inmediatamente saco el producto del camión (ya que si se llegó acá es que terminó la rama de la llamada a bt , y en la próxima iteración del ciclo se intentará poner el producto en el próximo camión si hay alguno).
- Si se llega acá significa que el producto no pudo meterse en ningún camión, entonces antes de agregar un camión nuevo y meter al producto ahí, calculamos si haciendo éso estamos empatando la cantidad de camiones que tiene nuestra mejor solución temporal. Si fuera así, no tiene sentido seguir. Si no, se agrega un camión con el p adentro, se llama a $bt(p+1, \text{camiones}, \text{mejorTemp}, M, n)$, e inmediatamente se saca al camión que habíamos agregado (ya que aquí ya terminamos de recorrer esa rama del árbol de backtracking)
- Llegado acá, la función simplemente termina.

La primera llamada a la función bt se hace con el producto 1, camiones tiene un camión con el producto 0, y siendo mejorTemp la solución trivial de n camiones con un producto en cada uno.

3.2.2. Pseudocódigo

3.3. Demostración

3.4. Análisis de complejidad

3.5. Test de complejidad

3.6. Testing